NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

MINIMIZATION OF MULTIPLE-VALUED
PROGRAMMABLE LOGIC ARRAY
USING SIMULATED ANNEALING

by

Robert C. Earle

December, 1991

Thesis Advisor: Jon T. Butler

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (*If applicable*) 32	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL (*If applicable*)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (*City, State, and ZIP Code*)	10 SOURCE OF FUNDING NUMBERS Program Element No / Project No / Task No / Work Unit Accession Number

11 TITLE (*Include Security Classification*)
Minimization of Multiple-Valued Programmable Logic Array Using Simulated Annealing

12 PERSONAL AUTHOR(S) Earle, Robert Charles

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (*year, month, day*)	15 PAGE COUNT
Master's Thesis	From To	December 1991	67

16 SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES			18 SUBJECT TERMS (*continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUBGROUP	Multi-Valued Logic; Minimization Heuristics; PLA Design; Simulated Annealing

19 ABSTRACT (*continue on reverse if necessary and identify by block number*)

Guaranteed minimal realizations of multi-valued programmable logic arrays can only be accomplished by an exhaustive search. Exhaustive search is not very realistic for complex expressions due to the immense amount of CPU time required to reach a solution. To circumvent this problem, heuristics have been developed. They provide near-minimal solutions, but use substantially less CPU time. This thesis investigates a new type of heuristic which is built on the foundation of simple implicant operations controlled by an annealing process. This new type of heuristic is superior to existing heuristics with respect to minimization capability but requires more CPU time.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL Butler, Jon T.	22b TELEPHONE (*Include Area code*) 408 646-3299	22c OFFICE SYMBOL EC/Bu

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

Approved for public release; distribution is unlimited.

Minimization of Multiple-Valued Programmable Logic Array Using Simulated Annealing

by

Robert C. Earle
Lieutenant, United States Navy
B.S., United States Naval Academy, 1985

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1991

ABSTRACT

Guaranteed minimal realizations of multi-valued programmable logic arrays can only be accomplished by an exhaustive search. Exhaustive search is not very realistic for complex expressions due to the immense amount of CPU time required to reach a solution. To circumvent this problem, heuristics have been developed. They provide near-minimal solutions, but use substantially less CPU time. This thesis investigates a new type of heuristic which is built on the foundation of simple implicant operations controlled by an annealing process. This new type of heuristic is superior to existing heuristics with respect to minimization capability but requires more CPU time.

TABLE OF CONTENTS

TABLE OF FIGURES

ACKNOWLEDGEMENT

I would like to express my sincere appreciation to the United States Navy for providing this exceptional educational opportunity. Special thanks go to Dr. Butler, Dr. Dueck, and Dr. Yang for their indispensable assistance. I would also like to thank my wife, Andy, for her extreme patience and support.

I. INTRODUCTION

A. BACKGROUND

The desire to use programmable logic arrays (PLA's) in circuit design is driven by their uniform structure and ease of implementation. If a PLA can be minimized, the amount of circuitry required is reduced, allowing for smaller size and increased performance.

The only existing guaranteed means of finding a minimal sum-of-products expression used in a PLA is by exhaustive search. This technique requires an extensive amount of execution time. An alternative solution is the use of heuristics. Such heuristics require significantly less computation time, while in most cases achieving a solution that is close to minimal.

Pomper & Armstrong [Ref. 1], *Dueck & Miller* [Ref. 2], and *Yang & Wang* [Ref. 3] are three example heuristics for minimizing sum-of-products expressions. The basis for their minimization technique is the direct cover method. Direct cover involves selecting 1) a minterm and 2) an implicant that covers that minterm. This sequence of events is duplicated until the expression is covered.

B. NEW APPROACH

An alternative approach is to manipulate existing implicants instead of creating them as is done in the direct cover method. The manipulation is carried out by basic operations such as combine, divide, and reshape. Random implicants are chosen and then checked to see if they can first be combined. If not, they are either reshaped or divided and sent back to the expression.

The divide and reshape operations actually allow for the number of implicants in the expression set to increase. This is a necessary evil in order to ensure that the heuristic has a chance to escape from a local minimum. A method of controlling this growth is needed.

Simulated annealing [Ref. 4] provides the necessary regulation by controlling the probability that a divide or a reshape operation (which increases the solution set) can occur. As with the mechanical applications of annealing, the solution set is heated and then allowed to slowly cool in order to reach a crystalline state or optimum solution. When the temperature is high, the probability of accepting a reshape or a divide operation is high (near 1.0). When the temperature is low, this probability is also low (near 0.0).

II. TWO NEW HEURISTICS

The heuristics *Cut & Combine* and *Reshape* are a result of the consolidation of the annealing process and the basic operations. The annealing process controls the use of the basic operations, while the basic operations manipulate the chosen pair of implicants. In order to develop an understanding of each heuristic, an examination of the annealing process and the mechanics of the operations is provided.

A. THE ANNEALING PROCESS

The simulated annealing process is used as a control function for the operations carried out by the two new heuristics. Each operation has a cost associated with it. The cost of a solution set is the number of implicants in it. If the number of implicants increases, then the operation results in an increase in cost. This occurs with the divide and reshape operations. If the number of implicants decreases, then a reduction in cost occurs.

The equation for determining the probability of accepting a cost increasing operation is

$$P(\Delta E) = \begin{cases} e^{\frac{-\Delta E}{k_B T}} \; if \; 0 \leq \Delta E \\ 1.0 \quad otherwise \end{cases}$$

ΔE represents the change in cost, as the result of executing the operation, while k_B is the Boltzman constant [Ref. 4] and T is the temperature. Initially, at high temperatures, $P(\Delta E)$ is high and all operations are allowed. After many moves under this condition, the solution set is significantly altered (i.e. melted). As the temperature cools, $P(\Delta E)$ decreases and there is less chance of accepting a cost increasing move. As the probability of accepting a cost increasing operation becomes extremely small, and few cost decreasing operations are possible, the solution set is considered to be frozen.

A schedule for the reduction of the temperature is needed. The temperature schedule used is

$$T_n = \alpha T_{n-1}.$$

The succeeding temperature is a fraction of the one before it. The typical range of α is between 0.80 and 0.99. For such values of α, the progression from melted to frozen state is very slow. Rapid cooling, quenching, has a fast execution time but does not produce favorable results. As with the actual mechanical characteristics of annealing, slow cooling provides the best results.

B. OPERATIONS

All the operations involve two implicants from a given function. A function can be defined in the following manner. An r-valued function, $f(x_1, x_2, \ldots, x_n)$, takes on a value from $\{0,1,\ldots,r-1\}$, for each assignment of values to the variables, which are also r-valued (i.e. $x_i \in \{0,1,\ldots,r-1\}$) [Ref. 7]. The radix, r, represents the number of logic values in the system. Due to the ease of implementation in multiple-valued PLA's, the truncated sum of the sum-of-products form of the function is used. An implicant or product term can be expressed as

$$c^{a1}x_1^{b1\ a2}x_2^{b2}\ldots^{an}x_n^{bn} \qquad (1)$$

where $c \in \{1,2,\ldots,r-1\}$, is a nonzero constant, where the literal function [Ref. 7],

$${}^{ai}x_i^{bi} = \begin{cases} r-1 & \text{if } ai \le x_i \le bi \\ 0 & \text{otherwise} \end{cases}$$

and where concatenation is the *min* function (i.e. $x\ y = \min(x,y)$). The literal function ${}^{ai}x_i^{bi}$ can only be the values 0 and r-1, the product (*min*) of laterals ((1) without c) is either 0 or r-1, while the complete term (with c) takes on values 0 and c. (1) is c if and only if $ai \le x_i \le bi$, for all i.

1. COMBINABILITY

Definition -- Two implicants:

$$I = c^{a1}x_1^{b1\ a2}x_2^{b2}\ldots^{an}x_n^{bn}$$

$$I' = c'^{a1'}x_1^{b1'\ a2'}x_2^{b2'}\ldots^{an'}x_n^{bn'}$$

are combinable if and only if c = c', ai = ai', and bi = bi' for all i ε {1,2,...,n} except for at most one i=j where bj+1 = aj' or bj'+1 = aj. If ai = ai' and bi = bi' for all i ε {1,2,...,n}, then I and I' are also combinable; in this case c does not have to equal c'.

In order to combine two implicants with 'n' variables, it is necessary and sufficient to have 'n-1' indices match. An index match occurs when the indices for common variables between implicant 1 and implicant 2 are exactly the same. The remaining variable must have indices from implicant 1 that abut with the indices from implicant 2. An example of abut as described in the definition above would be ${}^{0}x_1{}^{1}$ and ${}^{2}x_1{}^{3}$. The resulting implicant from the combining operation would be in one of two forms. ***a*** indicates the result of two implicants that are combinable and one implicant does not completely cover the other. ***b*** indicates the result of two implicants that are combinable and one implicant does completely cover the other. This operation decreases the total number of implicants by one. Implicant pairs 5-6 and 7-8 of Figure 2.1 provides an example of implicants that can be combined.

a) $I = c^{a1}x_1{}^{b1}{}^{a2}x_2{}^{b2}\ldots{}^{aj}x_n{}^{bj}$

b) $I = (c+c')^{a1}x_1{}^{b1}{}^{a2}x_2{}^{b2}\ldots{}^{an}x_n{}^{bn}$

2. **ABSORBABILITY**

Absorbability can be considered a subset of the combine operations.

Definition -- For two implicants:

$$I = c^{a1}x_1^{b1\ a2}x_2^{b2}\ldots^{an}x_n^{bn}$$

$$I' = c'^{a1'}x_1^{b1'\ a2'}x_2^{b2'}\ldots^{an'}x_n^{bn'}$$

I can be absorbed into I′ if and only if c′ = radix - 1, ai ≥ ai′, and bi ≤ bi′ for all i ε {1,2,...,n}.

If implicant I′ absorbs I, then I is no larger than I′, no part of I is not covered by I′, and the coefficient of I′ is radix-1. This operation also decreases the total number of implicants by one. For example, implicant 4 of Figure 2.1 can be absorbed into implicant 3.

3. **DIVISION**

Definition -- One implicant:

$$I = c^{a1}x_1^{b1\ a2}x_2^{b2}\ldots^{an}x_n^{bn}$$

can be divided in ((c-1) + Σ(bi - ai)) for all i ε {1,2,...,n} unique ways, if (1 ≤ c ≤ (radix - 2)). If c = radix - 1, then the number of ways of dividing the implicant is maximized because of the many ways the truncated sum can form. Table 2.1 lists the different possibilities for coefficient division for saturated and unsaturated coefficients.

TABLE 2.1 Unsaturated/Saturated Coefficient Division Pairs

Coefficient	Unsaturated Division Pairs	Saturated Division Pairs
3	1,2	1,2 1,3 2,2 2,3 3,3

The other type of division is geometric. Geometric division affects the literals. A geometric division can occur within any of the variables of the implicant. Each possible division has the same probability of occurring when the division operation is used.

4. ADJACENCY

Adjacency is used to determine which pair of implicants are sent to the operations [Ref. 5]. If the implicants are not adjacent, there is no chance of combining them and the pair is not considered. Both heuristics choose the pair of implicants randomly to ensure that all the possibilities for melting and then recombining the solution set have the same probability of being chosen. If the randomly chosen pair are not adjacent, the program simply chooses another pair until it finds two that are.

Definition -- Two implicants:

$$I = c^{a1}x_1^{b1}\,{}^{a2}x_2^{b2}\ldots{}^{an}x_n^{bn}$$

$$I' = c'^{a1'}x_1^{b1'}\,{}^{a2'}x_2^{b2'}\ldots{}^{an'}x_n^{bn'}$$

are adjacent if and only if bi" $\geq$ ai", for all iε{1,2,...,n}, where a" = max(a,a') and b" = min(b,b'), except for at most one i=j, aj" = bj"+1.

Two implicants are adjacent if there are no gaps; that is, there is no variable x_i such that bi > ai' or bi' > ai. Therefore, there is a path among contiguous values such that one can traverse from one implicant to the other without being

outside both. It also follows that a third implicant can be formed that consists of part of I and I′. For example, in Figure 2.1 implicant pairs, 1-2, 3-4, 5-6, and 7-8 are adjacent.

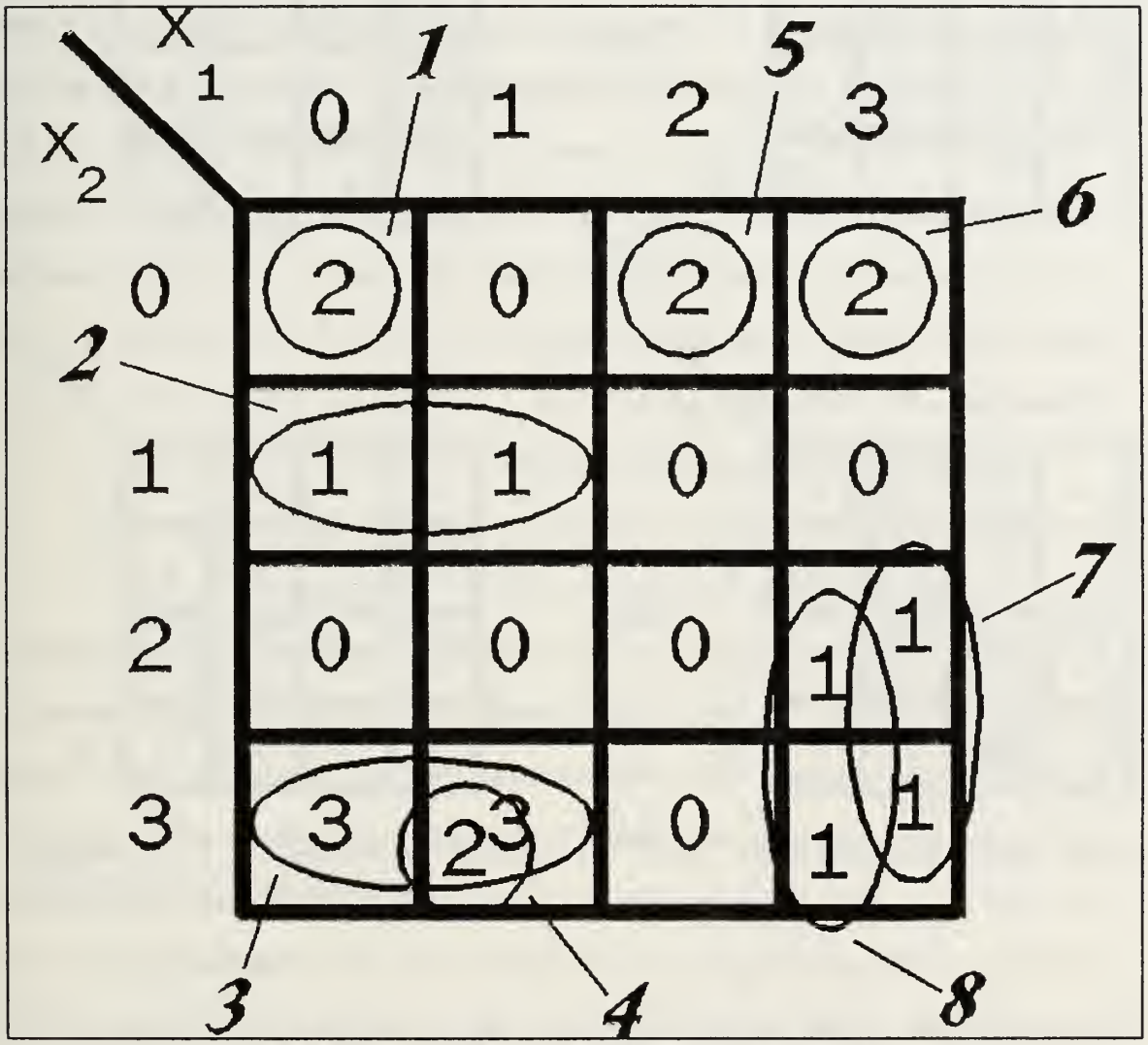

Figure 2.1: Example Showing Adjacent Pairs and Combination Candidates.

5. CONSENSUS

Consensus is a contributing function to the reshape operation. Consensus can be viewed as the common area or shared space between two implicants. When two implicants overlap, consensus is intersection of the two. If the terms do not overlap, then the consensus is a product term that is a part of both terms. The resulting coefficient is the minimum of the coefficient of the two product terms. Figure 2.2 illustrates this type of function best, with the hatched areas indicating the consensus. A formal definition of the consensus is presented as:

Definition -- Two implicants:

$$I = c^{a1}x_1^{b1}\,{}^{a2}x_2^{b2}\ldots{}^{an}x_n^{bn}$$

$$I' = c'^{a1'}x_1^{b1'}\,{}^{a2'}x_2^{b2'}\ldots{}^{an'}x_n^{bn'}$$

The consensus region is only present when the two implicants are adjacent. If the implicants overlap, as indicated in Figure 2.2a, then the consensus region is the area whose variable bounds are a" and b" for all $i\varepsilon\{1,2,\ldots,n\}$ where a" = max(a,a′) and b" = min(b,b′) and the resulting coefficient of the consensus region is c + c′. If the implicants do not overlap, as indicated in Figure 2.2b, then the region is the area whose variable bounds are a" and b" for all $i\varepsilon\{1,2,\ldots,n\}$ except for one i=j where b" = max(b,b′) and a" = min(a,a′) and the resulting coefficient of the consensus region is the minimum of c and c′.

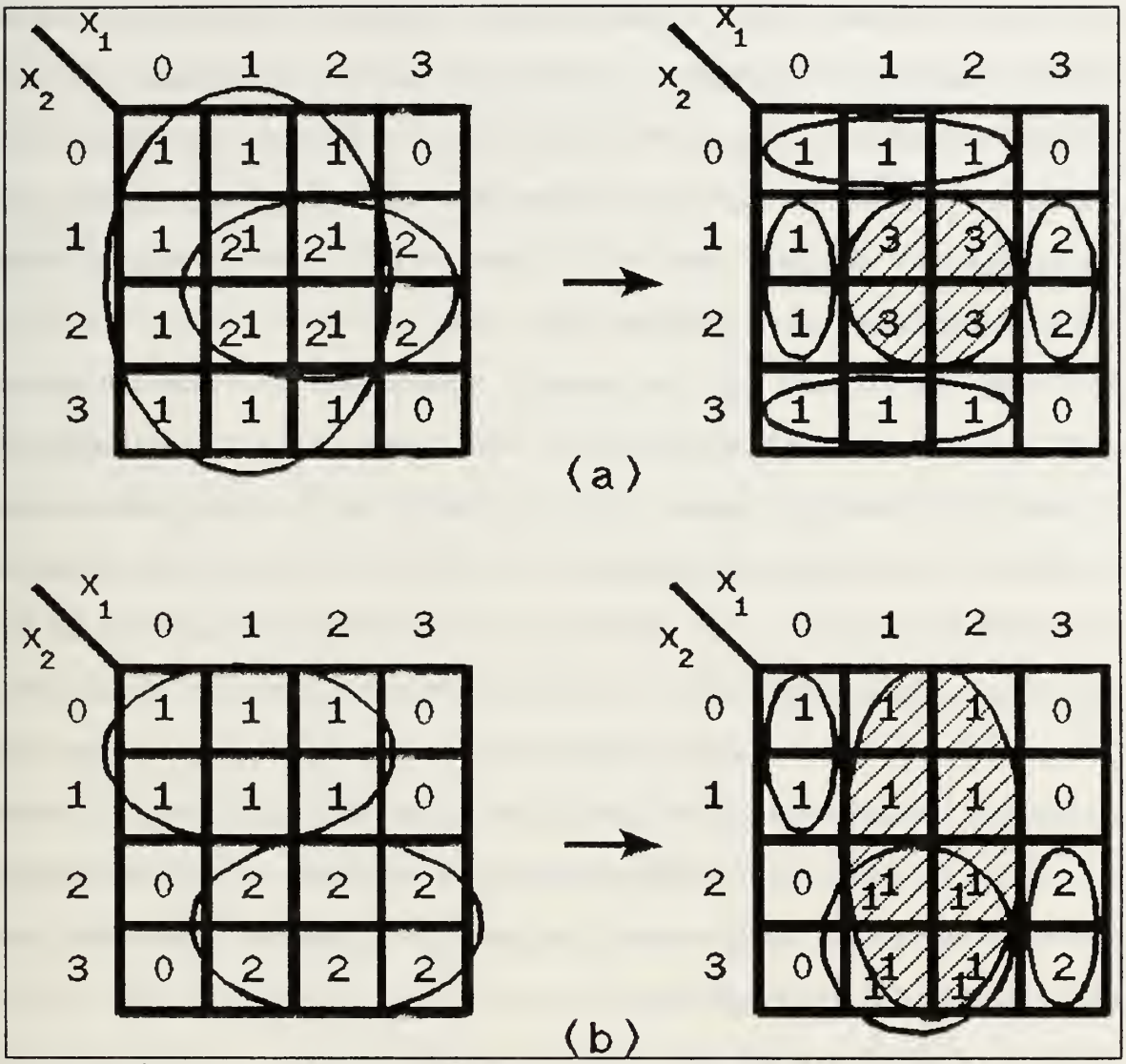

Figure 2.2: Consensus Examples for Overlapping and Non-overlapping Implicants.

6. RESHAPE

After the consensus of two non-combinable implicants is determined, the reshape operation will change or reshape the two implicants. As seen in Figure 2.2, the remaining implicants that have the consensus area removed from them must be divided. The remaining pieces of the implicant can be

divided in many ways. The reshape operation evaluates which divisions will form the lowest positive cost and then randomly chooses one of them for execution. Figure 2.2 gives an example of the division required after determining the consensus. Unlike the divide operation, reshape can produce a incremental cost greater than one.

The inability of the reshape operation to take advantage of saturated coefficients is a weakness. In some situations, treating the saturated coefficient as non-saturated can prevent an optimum solution from being found. For example, consider Figure 2.2a. The implicant formed as a result of the reshape operation has a coefficient of 3. In a radix 4 expression, the 3 is a saturated coefficient. Since the reshape operation does not allow for the coefficient to be treated as saturated, the resulting implicant with coefficient 3 will never be able to be divided as a saturated coefficient as described in Table 2.1.

C. HEURISTIC MECHANICS

1. *CUT & COMBINE*

The heuristics randomly chooses two implicants. If the pair are not adjacent, then the selection is continued until an adjacent pair is found. If possible, the two implicants of the chosen pair are combined. Otherwise, one of the adjacent pair is considered for the divide operation. If the divide operation is allowed in accordance with the probability of

accepting an operation that produces an increase in cost, one of the implicants of the pair is randomly chosen with each having an equal probability of selection. A list of all the possible divides is calculated for the chosen implicant. The divide operation then randomly chooses one of the possible divides and executes it, if possible. Minterms with coefficient 1 cannot be divided. In this case another adjacent pair are chosen and the cycle is repeated. Upon completion, the two new implicants formed by the division of the chosen implicant and the unchosen implicant are returned to the working space of the solution set. This sequence of events is repeated many times for each temperature. As the temperatures get cooler, the heuristic is less likely to accept cost increasing operations thus causing the solution set to migrate toward an optimum solution.

Simple divides performed by this heuristic are very fast, since the divisions are chosen randomly with no programmed intelligence. The negative aspect of the heuristic is that the randomness of the heuristic requires very slow cooling and a large number of operations per temperature to achieve optimum results.

2. ***RESHAPE***

The *Reshape* heuristic chooses random adjacent implicants and tries to combine them in the same manner as the *Cut &*

Combine heuristic. A major difference occurs when adjacent implicants are not combinable.

Reshape finds the consensus of the two implicants and then computes whether there will be a increase in the number of implicants in the solution set as a result of the operation. Since *Reshape* has operations which produce implicant size increases greater than one, the probability of accepting such a move is also dependent on how much the increase will be. At higher temperatures an acceptable cost increase may be as high as five. As the temperature decreases, the ceiling on acceptable cost increase is also lowered. As with *Cut & Combine*, the probability of increasing the cost is extremely low at low temperatures.

If the reshape operation has a cost increase which is accepted, then the consensus is taken and the implicants are reshaped and then returned to the solution set. This sequence is repeated many times for each temperature. The *Reshape* heuristic is more efficient due to the divide operations being chosen with programmed intelligence. A negative feature of the heuristic is the need to compute the consensus and positive cost for every non-combinable adjacent pair prior to determining if the operation will be accepted.

III. EXPERIMENTAL RESULTS

To evaluate these new heuristics, a 4 valued 4 variable function consisting of 200 randomly chosen minterms was selected. This expression was then minimized by the *Dueck & Miller* heuristic in the minimization program HAMLET [Ref. 2,6]. The resulting expression consisted of 87 implicants. This was then given to both *Cut & Combine* and *Reshape* to analyze the annealing process and the effect of varying program parameters.

A. ANNEALING PROCESS

Figures 3.1 and 3.2 show three-dimensional plots of the annealing process for *Cut & Combine* and *Reshape*. Horizontal "slices" represent a histogram of the distribution of visits to a solution set with the number of implicants given along the horizontal axis. At the initial temperature, it is apparent that the solution set is beginning to melt as the distribution of the solution set migrates toward solution sets containing higher numbers of implicants. At these relatively high temperatures, the annealing process allows a large number of operations which increase the cost. After the solution set reaches a melted equilibrium, around temperature = 0.68, the solution set then starts its migration toward an optimum solution.

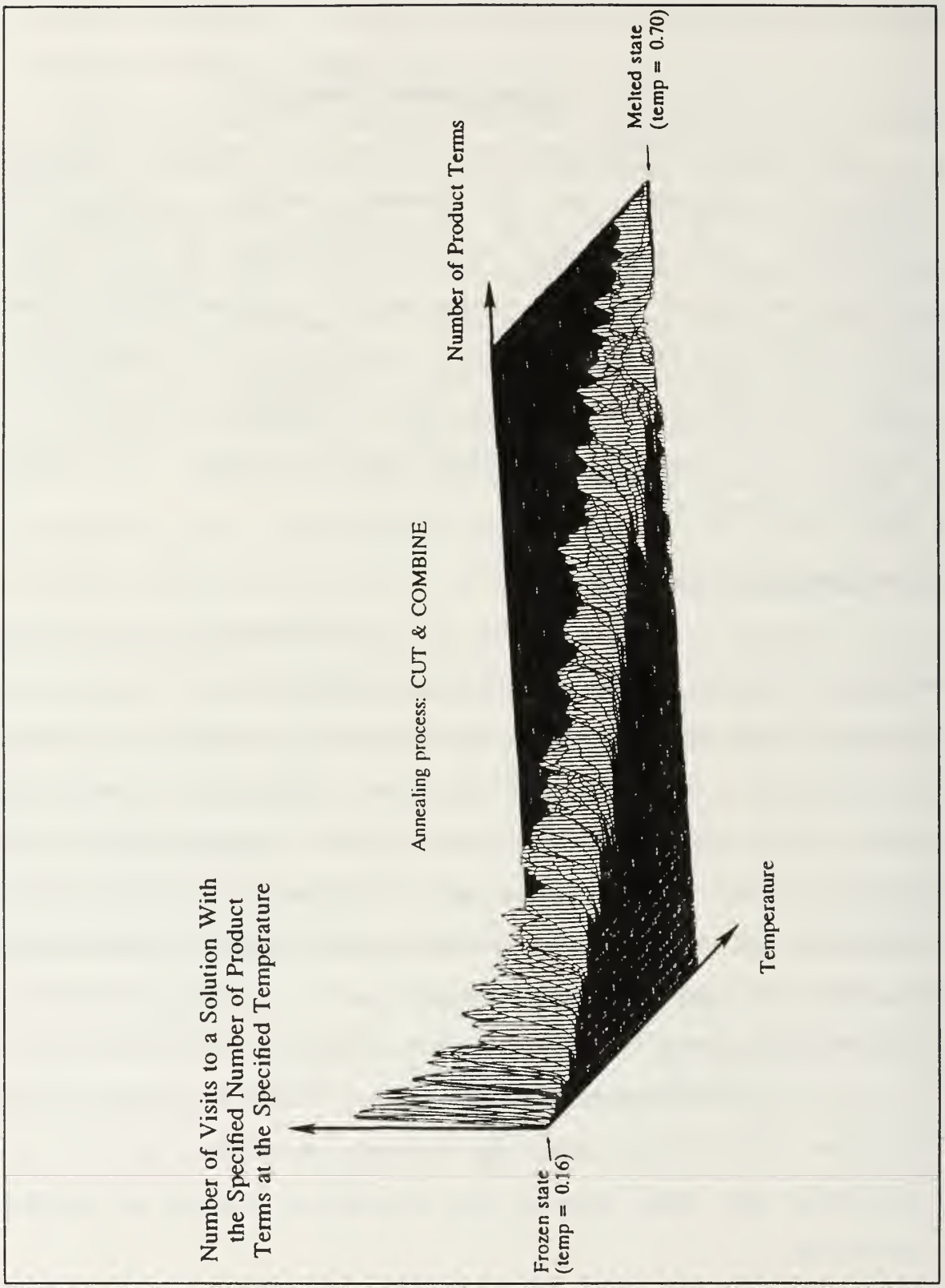

Figure 3.1: Simulated Annealing 3-D Plot for *Cut & Combine*.

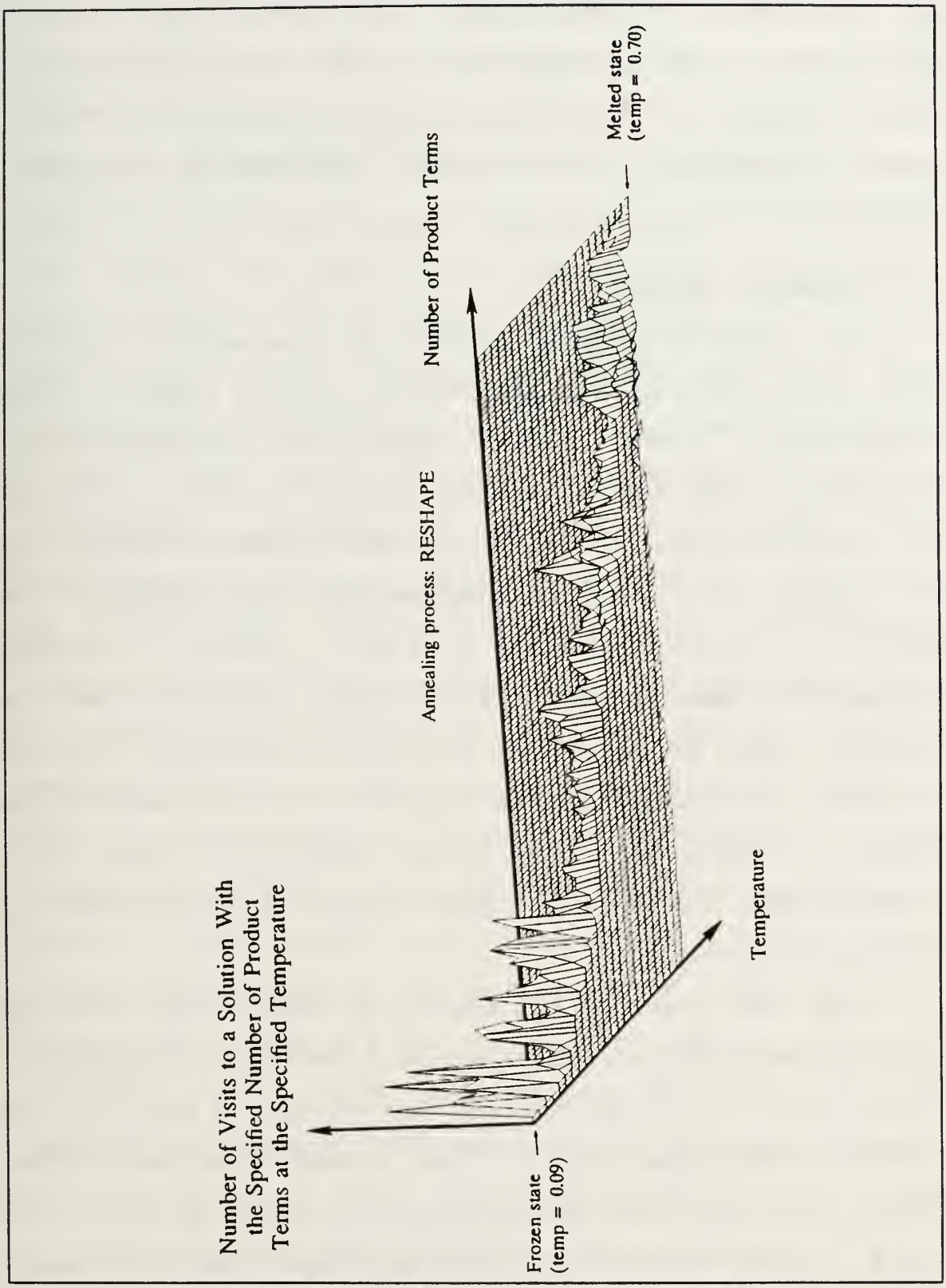

Figure 3.2: Simulated Annealing 3-D Plot for *Reshape*.

The number of temperatures each heuristic visits is controlled by the cooling rate, α. For *Cut & Combine*, $\alpha = 0.99$ is used, while for *Reshape*, $\alpha = 0.93$ is used. Since α is lower for *Reshape*, it visits fewer temperatures than *Cut & Combine*.

B. PARAMETER ANALYSIS

The annealing program requires six parameters: cooling rate (**cool rate**), maximum temperature (**initial temp**), minimum temperature (**lowest temp**), number of operations per temperature (**max moves**), maximum number of tries to achieve the required number of operations (**max attempted moves**), and the number of successive temperatures where reaching the maximum try count causes the program to proceed to the next temperature (**max frozen**). The program allows the user to directly input the parameters or rely on the default settings provided. The default parameter calculations are based on the number of minterms in the input expression. **Cool rate**, **initial temp**, and **max moves** have the most dramatic effect on program execution.

Cool rate controls the number of temperatures that the process sequences through during the transition from melted to frozen state. A higher cooling rate forces the heuristics to look at more temperatures, while a lower rate limits them. Very low cooling rates produce quenching, which is undesirable [Ref. 4], but significantly improve program speed. Extremely

high cooling rates cause the heuristics to run slowly (with marginally better results). Finding the optimum cooling rate (as low as possible while still able to achieve optimum results) is dependent on the operations implemented in the heuristic. This hypothesis is strengthened by the observation that *Reshape* and *Cut & Combine* perform best with different cooling rates. *Cut & Combine* operates best with a cooling rate equal to 0.99. This is due to the need for a slower reduction of the temperature necessary for the random process to have enough time to thoroughly melt the solution and then slowing cool the expression. *Reshape* performs best at 0.93. Its increased programmed intelligence enables it to be cooled at a faster rate. Figures 3.3 and 3.4 show the performance differences with varying cooling rates for both heuristics.

Initial temp controls the extent of melting. Since the temperature directly controls the probability of accepting a cost increasing operation, the higher the maximum or initial temperature of the process, the more melted the solution will become. Temperatures above 1.0 provided increased melting which did not improve the performance of either heuristic. Temperatures above 1.0 are not used to improve program cpu execution time. Lower temperatures would not allow the expression to properly melt thus making a new minimal solution unattainable. 0.7 is the optimum **initial temp** for both forms of the heuristics which correspond to a probability of 0.24 for accepting a positive cost move. Figure 3.3 and 3.4 show

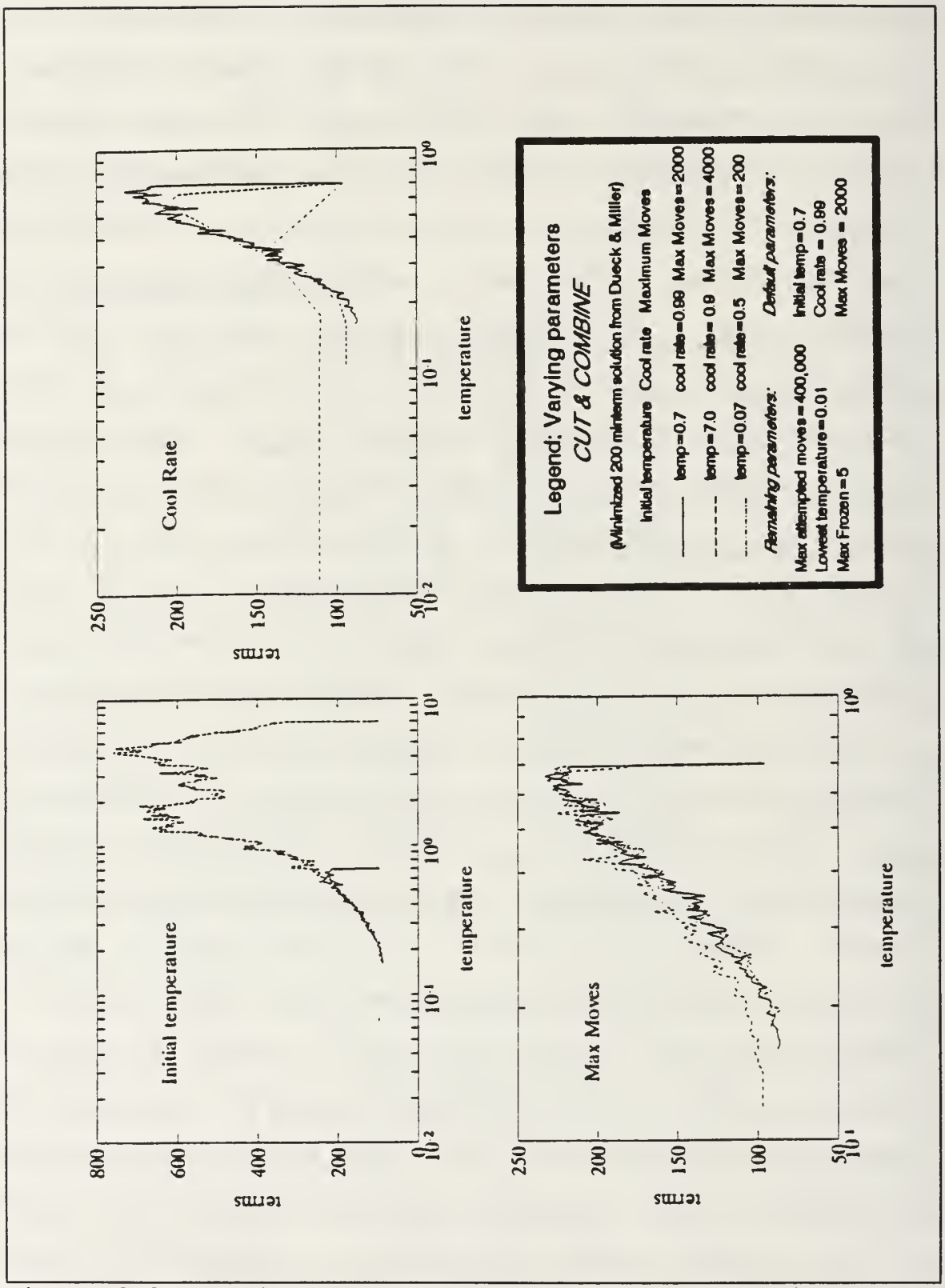

Figure 3.3: Varying Parameter Plots for *Cut & Combine*.

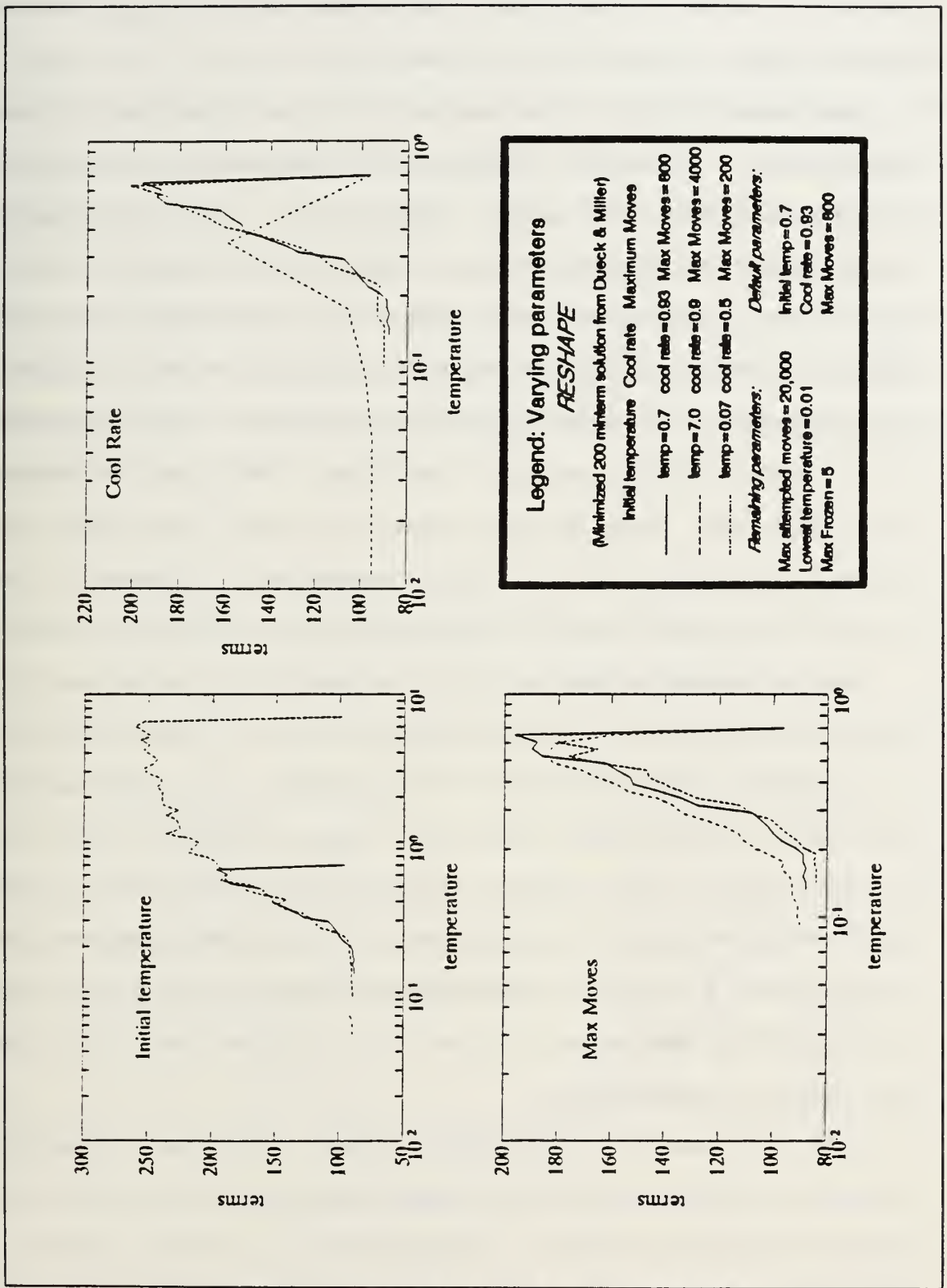

Figure 3.4: Varying Parameter Plots for *Reshape*.

the performance differences with varying initial temperatures for both heuristics.

Max moves control the number of moves occurring at each temperature. This is analogous to the amount of work or energy expended at each temperature. The programmed intelligence of the operations used by each heuristic has a significant effect on the number of operations required. *Reshape* does not require as much effort (four times the number of minterms in the input expression) because of the programmed intelligence of the reshape operation, while *Cut & Combine* performed best when the max moves is high (ten times the number of minterms in the input expression). Figure 3.3 and 3.4 show the performance differences with varying max moves.

Max attempted moves controls how hard the program works in the later stages of the annealing process. As the probability of accepting a positive cost move decreases, the program must take time to find a pair that will allow a negative cost move or operation. This number regulates how long the program examines moves prior to continuing on to the next temperature. *Reshape* uses a value for **max attempted moves** equal to 25 times the number of **max moves**, while *Cut & Combine* uses 210 times the number of **max moves**.

The program is terminated either when the annealing schedule reaches the minimum temperature specified or when the maximum attempted moves is reached for a certain number of successive temperatures. This number, **frozen count** is used to

minimize wasted computing effort. Five successive temperatures where **max attempted moves** is achieved indicates that the solution is frozen and that no more effort should be expended on the expression. In most cases, it is **frozen count** that terminates the program, with the minimum temperature seldom being reached prior to the expression being frozen. This phenomenon occurs because the minimum temperature is set very low to ensure that the program is allowed to reach a frozen state. *Reshape* and *Cut & Combine* operate with minimum temperatures equal to 0.01.

IV. COMPARISON OF NEW HEURISTICS WITH OTHER MINIMIZATION HEURISTICS

In order to present a fair comparison of the simulated annealing heuristics with the other established minimization heuristics, two groups of test functions are analyzed. The first group is selected for their unique characteristics, while the second group is selected randomly and without bias.

The first test set contains three test functions. Test1 is a 4 valued 3 variable function which originated as fifty randomly chosen minterms and was then minimized by the *Dueck & Miller* heuristic in HAMLET [Ref. 2,6]. The output from HAMLET provided a expression with 24 product terms or implicants. An exhaustive search in HAMLET concluded that the minimal solution contains 21 product terms.

To demonstrate a potential problem with the *Cut & Combine* heuristic, a 4 variable 4 valued symmetric function with a minimal solution consisting of six large implicants was chosen for Test2. The key point needed to validate the potential problem was that the six implicants of the minimal solution needed to be adjacent. In the melted state the larger implicants are broken down into smaller ones, thus destroying group integrity. As the temperature cools, new groups are formed which have different group boundaries than the actual minimal solution. This causes the heuristic to perform poorly since there is only one combination of groups that allow for

the minimal solution. If the minimal solution had been composed of smaller subsets of implicants allowing for different combination sequences to achieve the solution, then *Cut & Combine* would have been able to perform adequately.

Reshape does not experience this phenomenon to the same degree as *Cut & Combine* because the reshape operation preserves the initial group boundaries and attempts to build or add on to the group. The massive dissection used by *Cut & Combine* can be a positive effect, although in this case it hinders the performance of the *Cut & Combine* heuristic, as indicated in Table 4.1.

Test3 is a designed 4 valued 2 variable function which requires over-summing in order to achieve the minimal

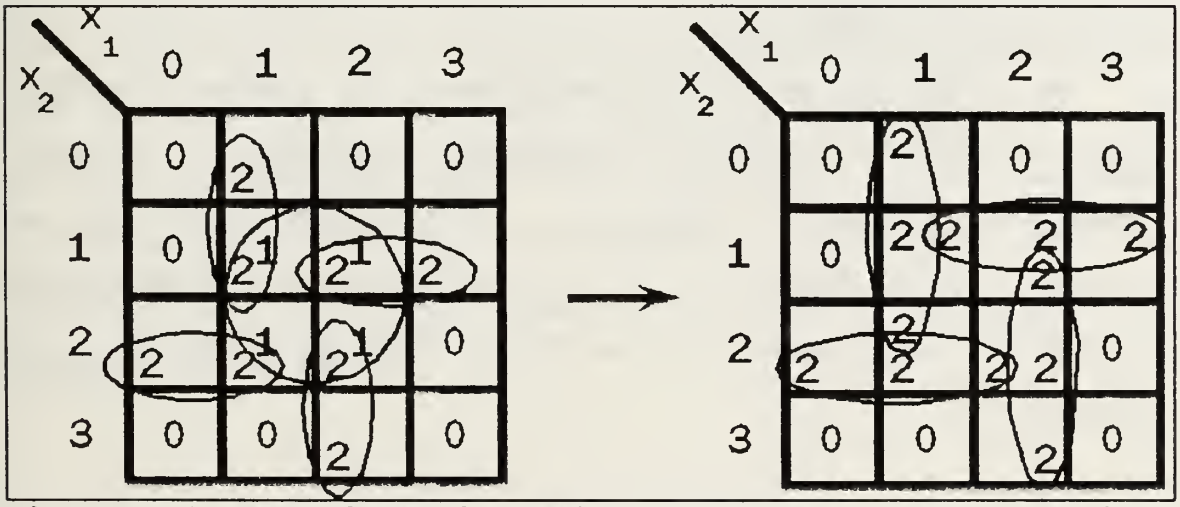

Figure 4.1: Map of Test3 Showing the Effect of Over-Summing.

solution. Figure 4.1 is a graphical representation of the input function.

This expression was chosen to analyze the performance of the *Reshape* heuristic which does not have an over-summing capability. The only heuristics able to find the minimal solution are *Cut & Combine* and *Yang & Wang* [13]. The performance of the heuristics is shown in Table 5.1. The number of product terms output by each heuristic and the cpu time required is shown. The performance of *Reshape* and *Cut & Combine* on test1 deserves attention due to their ability to find the minimum solution with extremely little effort. The cpu time is much greater for the *Cut & Combine* heuristic due to the increased effort required as a result of the lack of programmed intelligence in the divide operation as compared to the reshape operation.

The next group of test sets were all chosen randomly. Each random 4 valued 4 variable expression contains a specified number of minterms. Nine ensembles of ten were chosen over a range from 25 to 250 minterms per expression. Heuristic performance for the random test sets can be seen in Figure 4.2.

TABLE 4.1: Test Set One Comparisons

Heuristic	Test 1 in/out/time	Test 2 in/out/time	Test 3 in/out/time
Cut & Combine	24/21/6264	14/19/2125	5/4/207*
Reshape	24/21/147	14/7/71.0	5/5/4.6
Dueck & Miller	24/24/0.9	14/6/5.65	5/5/0.03
Pomper & Armstrong	24/24/0.4	14/10/2.98	5/5/0.01
Yang & Wang	24/22/8.6	14/10/9.32	5/4/0.12

*Found the minimal solution in 4.2 seconds; program completion in 207 seconds.

Cut & Combine performed best on the ensembles having fewer minterms, while *Reshape* had the best performance on the remaining functions. As with the selected test expressions, the required cpu time for execution is dramatically higher for the *Cut & Combine* heuristic as demonstrated in Figure 4.3. The overall results show that the increased intelligence of the *Reshape* heuristic not only improves performance but also CPU execution time. Both annealing heuristics, on the average, outperform the other heuristics. The increased performance is not without a price. CPU times for the annealing heuristics are higher than for the other heuristics implemented in HAMLET.

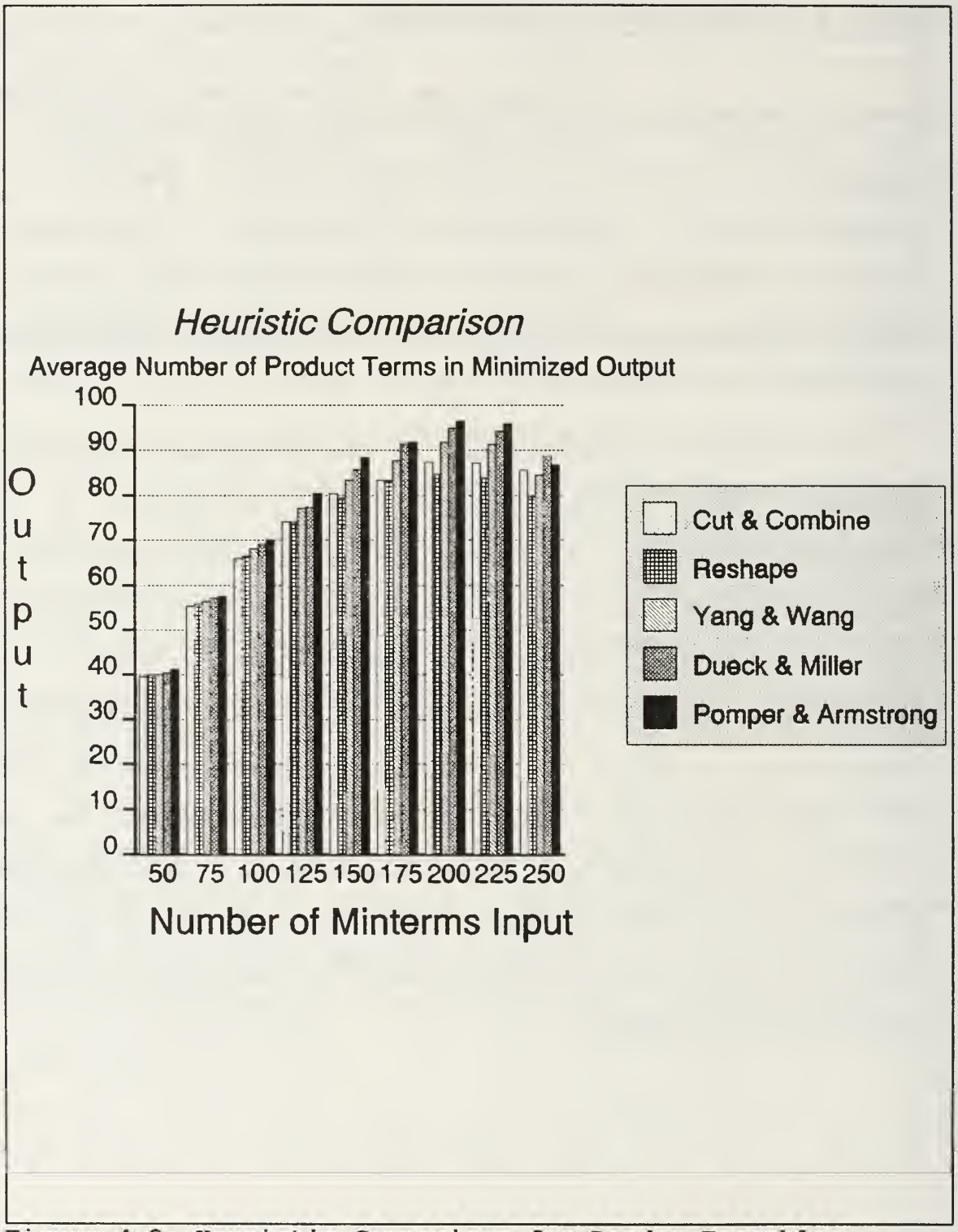

Figure 4.2: Heuristic Comparison for Random Ensembles.

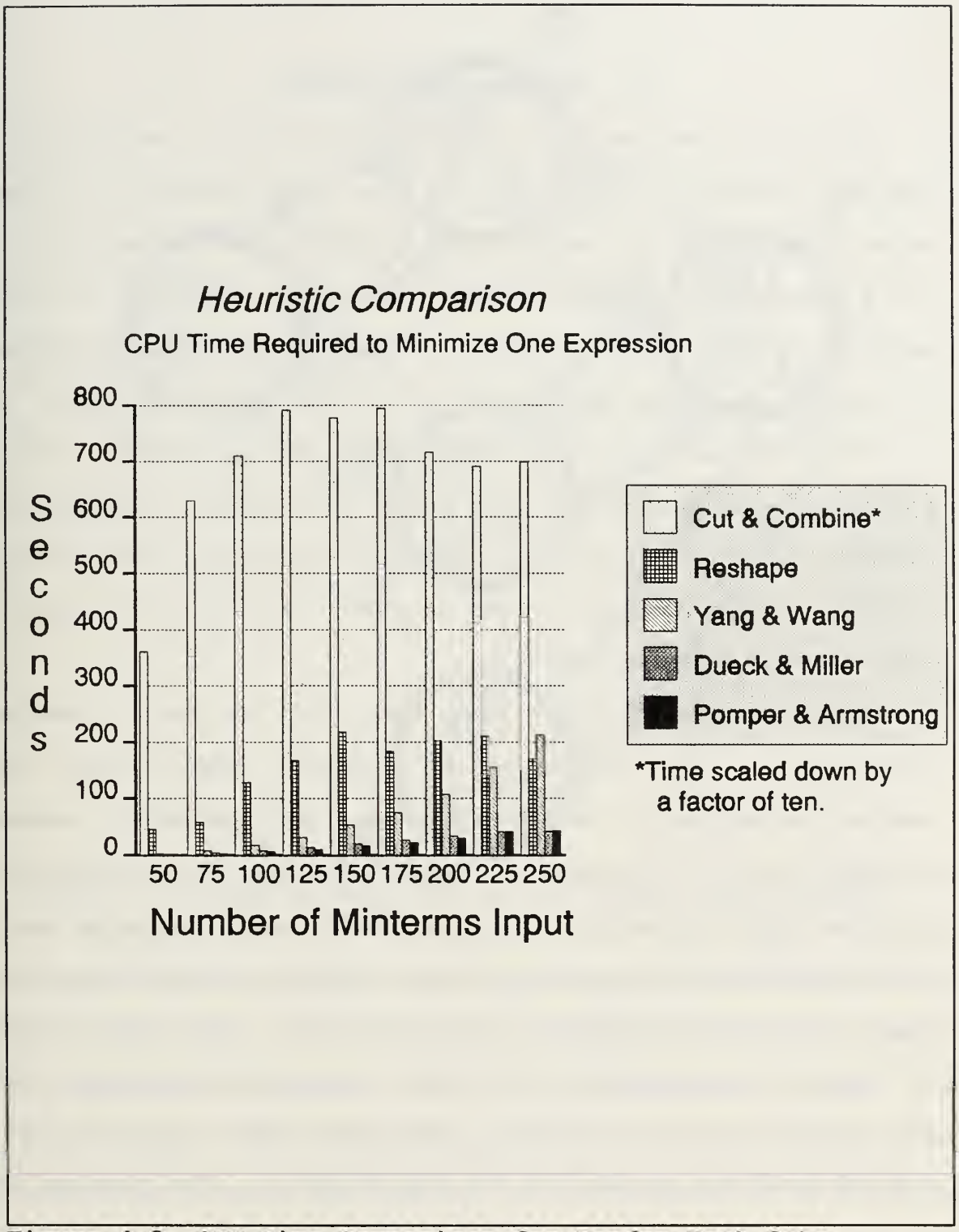

Figure 4.3: CPU Time Comparison for Random Test Sets.

V. PROGRAM VERIFICATION

Since the new heuristics are probabilistic, verification can be difficult. To aid in the understanding of the annealing heuristics and provide a type of verification, Dr. Jon T. Butler derived the Markov chain model described in this section. If a small problem is examined, the probabilities of state transition for the various states corresponding to the solution sets with different number of implicants. After running the program for a long time, experimental probability results can be derived for comparison with the theoretical results. This comparison can provide insight to the proper execution of the program.

The annealing process when used for the *Cut & Combine* heuristic can be thought of as a Markov chain, where the states correspond to the solution sets with different number of implicants. For analysis, observe the expression formed by the two implicants, 1 and 2 from Figure 2.1. There are only six implicants that cover this is expression, while there are only five ways of combining the implicant. The five circles of Figure 5.1 represent each of the unique combinations. The circle to the left, state 1, represents the unique minimal sum-of-products expression for the function. The circles in

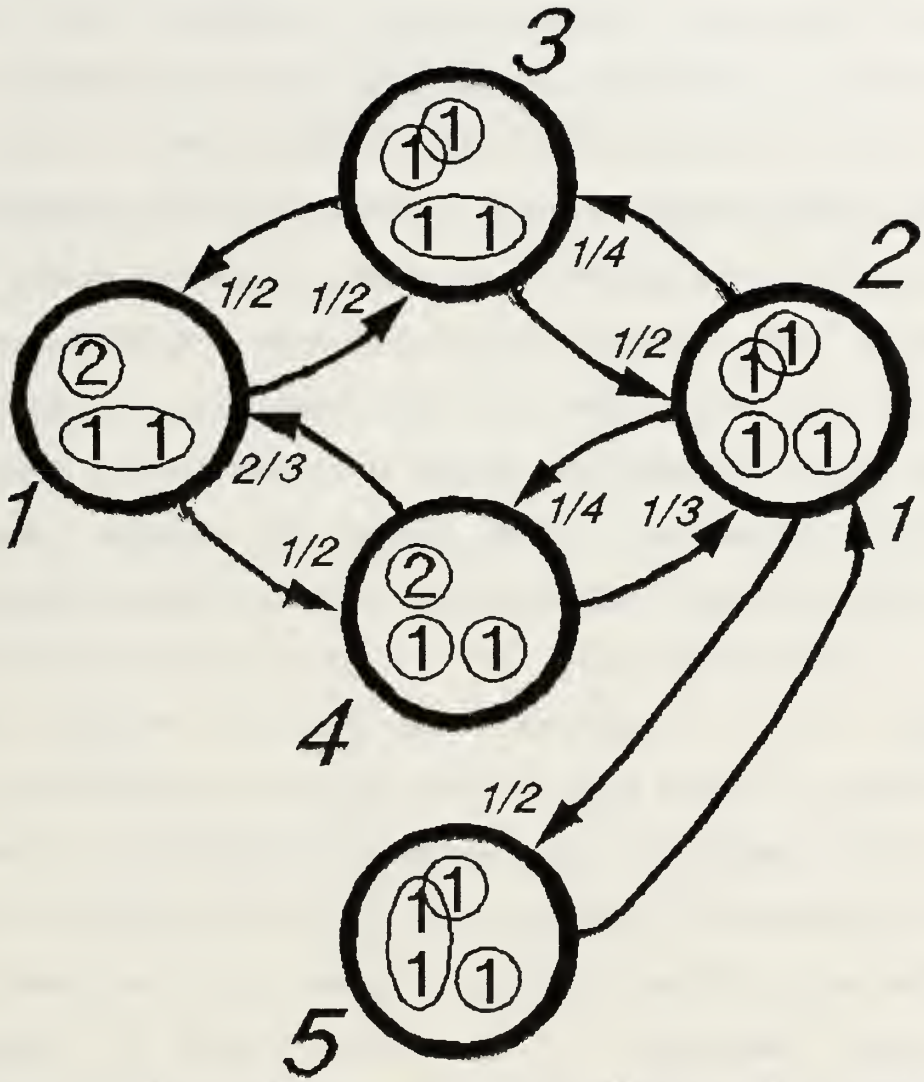

Figure 5.1: State Transitions in the Markov Chain Model.

the middle, states 3, 4, and 5, represent sum-of-product expressions each having three implicants in the solution set. The circle on the right, state 2, correspond to sum-of-product expressions having four implicants in the solution set. The arcs between the states correspond to the probability of transition between the states. If the program is in state 3, there are two possible transitions, one to state 1 and the

other to state 2. The implicants of state 1 are a single 2 and a pair of adjacent 1's. There are three possible choices for a pair to be selected from the implicants of the solution set associated with state 3. Each pair has a probability of 0.33 of being selected. In order to have a transition to state 1, the pair of single 1's must be chosen and then combine to form a single 2. A transition to state 2 will occur if any of the other pairs are chosen since they will be unable to combine. Being unable to combine, one of the implicants chosen will be picked with a probability of 0.5 and sent to the divide operation. If the implicant represented as a pair of 1's is chosen, they will be divided into two individual 1's and the solution set will transition to state 2. If the implicant represented by a single 1 is chosen, the divide operation is not possible since the single 1 can not be divided any further. Another adjacent pair is chosen and the procedure repeated. The equation used to compute the probability of transitioning from state 3 to state 1 is

$$\frac{\frac{1}{3}}{\frac{1}{3}+((2)(\frac{1}{3})(\frac{1}{2}))}=\frac{1}{2},$$

while the probability of transitioning from state 3 to state 2 is

$$\frac{2\left(\frac{1}{3}\right)\left(\frac{1}{2}\right)}{\frac{1}{2}+(2)\left(\frac{1}{3}\right)\left(\frac{1}{2}\right)}=\frac{1}{2}.$$

The remaining probabilities for state transition were computed in the same manner and are shown above each arc in Figure 5.1.

A more concise representation of the state transition probabilities is in the form of a stochastic matrix. The transition probability from state i to state j is the element p_{ij} of the matrix. The stochastic matrix for the expression represented in Figure 5.1, S is given as

$$S=\begin{vmatrix} 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{2}{3} & \frac{1}{3} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{vmatrix}.$$

Consider $P = [p_0, p_1, \ldots, p_n]$ to be a row matrix, where the initial probability of state i is represented by p_i. Then the probabilities after one transition are $P(S)$. Following the same principle, $P(S^k)$ shows the probabilities after k transitions. The matrix for S^{20} can be seen in equation (2). There is an apparent consistency in the final state. For analysis, consider the initial state being either 1 or 2. The probability of being in state 1 is 0.412 and being in state 2

$$S^{20}=\begin{vmatrix} 0.412 & 0.412 & 0 & 0 & 0 \\ 0.588 & 0.588 & 0 & 0 & 0 \\ 0 & 0 & 0.353 & 0.353 & 0.353 \\ 0 & 0 & 0.353 & 0.353 & 0.353 \\ 0 & 0 & 0.294 & 0.294 & 0.294 \end{vmatrix} \quad (2)$$

is 0.588 after 20 transitions. Now consider if the initial state is 3, 4, or 5, then 0.353, 0.353, or 0.294 are the probabilities of being in that state, respectfully. A common characteristic of Markov chains is their independence of the initial state. A formal analysis can be applied by taking the transformation of the matrix [Ref. 7]. S can be characterized as

$$S=L^{-1}\Lambda L,$$

where the rows of L are the left eigenvectors x_1, x_2, ... , x_m of S, corresponding to eigenvalues λ_1, λ_2, ... , λ_m, respectively, and Λ is a diagonal matrix with entries λ_1, λ_2, ... , λ_m [Ref. 5]. An expression for S^k is

$$S=L^{-1}\Lambda L=\sum_{i=0}^{m}\lambda_i^k y_i xi,$$

where x_i is the i-th column of L and y'_i is the i-th column of L_{-1} [Ref. 5]. The eigenvalues for S are determined to be

$$\lambda_1=\frac{\sqrt{42}}{12},\ \lambda_2=\frac{\sqrt{42}}{12},\ \lambda_3=-1,\ \lambda_4=1,\ \lambda_5=0.$$

Let $B \sim A^k$ represent

$$\lim_{n\to\infty} \frac{b}{a} = 1$$

for very element b in square matrix B and for the corresponding elements a in square matrix A^k. Then the following equation applies:

$$S \sim (-1)^k y_3 x_3 + y_4 x_4 .$$

y'_3, x_3, y'_4, and x_4 can be calculated from the eigenvectors of S. This computation produces two matrices. The first is for even values of k

$$S^k = \begin{vmatrix} \frac{14}{34} & \frac{14}{34} & 0 & 0 & 0 \\ \frac{14}{34} & \frac{14}{34} & 0 & 0 & 0 \\ 0 & 0 & \frac{12}{34} & \frac{12}{34} & \frac{10}{34} \\ 0 & 0 & \frac{12}{34} & \frac{12}{34} & \frac{10}{34} \\ 0 & 0 & \frac{12}{34} & \frac{12}{34} & \frac{10}{34} \end{vmatrix} .$$

The second is for odd values of k as seen in equation (3).The probability of choosing an even or odd value of k is 0.50. Based on this fact the probabilities of the specific states can be computed and are listed in Table 5.1.

$$S^k = \begin{vmatrix} 0 & 0 & \frac{12}{34} & \frac{12}{34} & \frac{10}{34} \\ 0 & 0 & \frac{12}{34} & \frac{12}{34} & \frac{10}{34} \\ \frac{14}{34} & \frac{14}{34} & 0 & 0 & 0 \\ \frac{14}{34} & \frac{14}{34} & 0 & 0 & 0 \\ \frac{14}{34} & \frac{14}{34} & 0 & 0 & 0 \end{vmatrix} \qquad (3)$$

TABLE 5.1: Probability of Various States After Many Transitions

State	Probability of the State
1	0.206
2	0.294
3	0.176
4	0.176
5	0.147

This table demonstrates that while in the melted state the probability of being in the minimal state is about 20%. This phenomenon occurs because when the temperature is high and the solution set is melted and the annealing process is more likely to accept positive cost operations. An interesting fact is that the probability of being in the false minimum, state 5, is less than the probability of being in the true minimum, state 1.

The above analysis is for the highest temperature, when all operations including positive cost ones are permitted. At the lower temperatures, positive cost moves are accepted with the probability

$$P(\Delta E) = e^{\frac{-\Delta E}{k_b T}}$$

For the heuristic *Cut & Combine*, the highest cost increase an operation can have is 1. The probability of accepting this type of operation (divide) is p. The corresponding transition matrix is

$$S_p = \begin{vmatrix} 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \\ \frac{\frac{1}{2}}{\frac{1}{2}+\frac{1}{2}p} & \frac{\frac{1}{2}}{\frac{1}{2}+\frac{1}{2}p} & 0 & 0 & 0 \\ \frac{\frac{2}{3}}{\frac{2}{3}+\frac{1}{3}p} & \frac{\frac{1}{3}}{\frac{2}{3}+\frac{1}{3}p} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{vmatrix}$$

Using the same type of analysis, the probabilities that the system is in a specific state can be computed to ascertain their reliance on p. Table 5.2 presents the probability of each state based on p. It is apparent the as p decreases, the probability of state 1 approaches 0.5, while the probability

of state 5 decreases to 0.0. The probability of any one state cannot exceed 0.5 because a transition is required after each state. As p decreases the probability of state 1 (minimum state) goes up, while the probability of state 5 (false minimum state) goes down. This effect is quite desirable and is a significant plus for the annealing heuristics.

TABLE 5.2: Probability of Various States After Many Transitions as a Function of p, the Probability of Accepting a Positive Cost Operation

p	state 1	state 2	state 3	state 4	state 5
1.0	0.206	0.294	0.176	0.176	0.147
0.5	0.289	0.211	0.197	0.197	0.105
0.250	0.365	0.135	0.217	0.217	0.067
0.125	0.422	0.078	0.239	0.239	0.039
0.06250	0.457	0.043	0.239	0.239	0.021
0.03125	0.478	0.022	0.244	0.244	0.012
0.01562	0.489	0.011	0.247	0.247	0.005
0.00781	0.494	0.006	0.249	0.249	0.003

VI. CONCLUSIONS

The results and analysis of the test runs show significant promise for the annealing heuristics. These heuristics show better performance, in general, than their direct-cover counterparts.

The ability of the user to directly modify the heuristic control parameters provides a flexibility to allow the heuristics to perform well with many types of minimization problems. This flexibility also has a cost. Although the program automatically provides a reasonable set of parameters, optimum performance can only be achieved by an iterative technique for each function. Again, a time verses performance type of problem arises. If the user has the time and resources to dedicate to finding a near minimum solution, these heuristics have the capability of doing so.

Cut & Combine was the first annealing heuristic developed and can be considered the foundation in which *Reshape* was built upon. The operations were simple and performed well with a reasonable amount of effort. In an attempt to further streamline the heuristic, Dr. Gerhard Dueck developed the reshape operation. The increased programmed intelligence decreased CPU time execution and improved heuristic performance. Although functions can be found to accentuate

the weaknesses of each, the *Reshape* heuristic proved to have the best overall performance of any of the tested heuristics.

It is theorized that as the operations of annealing heuristics are further streamlined or gain more programmed intelligence, their performance will continue to improve. Further research should include modifying the existing reshape operation to compensate for over-summing and trying to implement the no rejected operation principle [Ref. 8].

These family of heuristics have a great deal of potential and can provide yet another path in the quest for multi-valued logic array minimization.

VII. APPENDIX: C CODE UTILIZED

Enclosed in this appendix are the two C programs for the *Cut & Combine* and *Reshape* heuristics in conjunction with HAMLET [Ref. 6]. Each program contains routines that are used by both heuristics.

1. C code for annealing control:

```
/* $Source: cc.c $
 * $Revision: 1.0 $
 * $Date: 91/09/05 23:37:28 $
 * $Author: Earle and Dueck $
 * "modifications to original program of yurchak"
 */

/* $Log: cc.c $
 * Revision 2.0  91/09/05 23:37:28  earle/dueck
 * Original version
 */

/* cc.c

    - This module controls the annealing process for both
heuristics; Cut & Combine and Reshape.  Initially written just
for the Cut & Combine Heuristic with a later modification to
include the Reshape Heuristic.  Additional routines used for
testing and analysis are included.
*/

#include "defs.h"
#include <math.h>
double a_temp,enumber[MAX_TAB];
int valid = 0;
extern int R_flag;

Cut_Combine()
 {
    int       i,j,min_term,absolute_min,max_term,cost;
    int        sum,frozen_count = 0,try_count;
    int       *X;
    Implicant  *I;
```

```
    extern double   min_temp,max_temp,cool_rate;
    extern int      max_frozen, max_try_count;
     extern    int  max_valid_count;
    long   init_cpu,clock();
    double tot_cpu;
    FILE        *fp;

/* Open output statistics file and initial clock counter for
   Cpu time comparisons. */

    fp = fopen("stats.out","a");
    init_cpu = clock();
    if (E_final[C_C].I != NULL)
        dealloc_expr(&E_final[C_C]);

    HEUR = C_C;
    dup_expr(&E_work,&E_orig);

    /* set parameters */

    if (R_flag)
       {
       if (max_valid_count == 0)
          max_valid_count= minterms()*MAX_VALID_FACTOR_R;
       if (max_try_count == 0)
          max_try_count = max_valid_count*MAX_TRY_FACTOR_R;
        if (cool_rate == 0.0)
          cool_rate = COOL_RATE_R;
     }
    else
       {
       if (max_valid_count == 0)
        max_valid_count = minterms()*MAX_VALID_FACTOR_C;
       if (max_try_count == 0)
        max_try_count = max_valid_count*MAX_TRY_FACTOR_C;
       if (cool_rate == 0.0)
        cool_rate = COOL_RATE_C;
     }

    if ((E_work.I=(Implicant*)
     realloc(E_work.I,sizeof(Implicant)*MAX_TERM))==NULL)
        fatal("alloc_implicant(): out of memory\n");
    if (!vverify()) printf("we are in big trouble !!!\n");

    E_final[HEUR].nterm = 0;
    E_final[HEUR].radix = E_orig.radix;
    E_final[HEUR].nvar = E_orig.nvar;
    E_final[HEUR].I = NULL;
```

```
    resource_used(START);
    tot_cpu = 0.0;
    absolute_min = 1000000;
    a_temp = max_temp;

    printf("max_temp = %5.2f cool_rate = %5.3f min_temp =
                %5.3f\nmax_frozen = %d ",
               max_temp,cool_rate,min_temp,max_frozen);
    printf("max_try_count = %d max_valid_count = %d\n",
              max_try_count,max_valid_count);

/* Temperature control mechanism */

    while ((a_temp > min_temp) &&
                (frozen_count < max_frozen))
         {
        valid = 0;
        max_term = 0;
        min_term = 1000000;
        try_count = 0;
        for(i = 1; i < MAX_TAB; i++)
        enumber[i] = exp(-i/a_temp);
        enumber[0] = exp(-0.05/a_temp);

/* Count control mechanism */

        while((try_count < max_try_count)&&
               (valid < max_valid_count))
               {
            make_a_move();
                  if (E_work.nterm< min_term)
                min_term = E_work.nterm;
            if (E_work.nterm> max_term)
               max_term = E_work.nterm;
                try_count++;
             }

        if (absolute_min > min_term)
             absolute_min = min_term;
        if ((valid >= max_valid_count) && (min_term <
              max_term))
            frozen_count = 0;
        else
            frozen_count++;

/* Cpu time computation */

        tot_cpu = tot_cpu + (clock() - init_cpu)/1000.0;
```

```
        init_cpu = clock();

          printf(" %7.3f",a_temp);
        printf(" %3d %3d %3d %6d",
                   min_term,E_work.nterm,max_term,try_count);
        printf(" %10.3f\n",tot_cpu/1000.0);
        a_temp = cool_rate  * a_temp;
    }

    resource_used(STOP);

    dup_expr(&(E_final[C_C]),&E_work);
    dealloc_expr(&E_work);

    tot_cpu = tot_cpu + (clock() - init_cpu)/1000.0;

    printf("cpu time used = %10.3f sec.\n",tot_cpu/1000.0);
    fprintf(fp,"%5.3f %4d %4d %10.3f\n",
             cool_rate,E_orig.nterm,
            absolute_min,tot_cpu/1000.0);
    fclose(fp);

}

print_map2()
/*************************************************************
function:
   Print the Karnaugh map of E_work in its present state
*************************************************************/
{
    register   i,j;
    int X[MAX_VAR+2];
    int *V;

    for (i=0; i < nvar; i++) X[i] = 0;
    for (i=0; i < nvar;)
       {
        V = eval(&E_work,X);
        printf("%s%3d%c",X[i]==0?"
                  ":"",V[EVAL],V[HLV]?'.':' ');
        X[i]++;
        for (;i < nvar;)
          {
           if (X[i] >= radix)
               {
               X[i] = 0;
               i++;
               X[i]++;
```

```
                }
            else
                {
                i = 0;
                break;
                }
          }
      }
    printf("\n");
}


int vverify()
/*********************************************************
function:
   Verify that the integrity of the function is maintained
   during the program execution.
*********************************************************/
{
    register   i,j;
    int X[MAX_VAR+2];
    int *V;
    int    first,second;

    for (i=0; i < nvar; i++) X[i] = 0;
    for (i=0; i < nvar;) {
        V = eval(&E_work,X);
        first = V[EVAL];
        V = eval(&E_orig,X);
        second = V[EVAL];
        if (first != second) return(0);
        X[i]++;
        for (;i < nvar;) {
           if (X[i] >= radix) {
               X[i] = 0;
               i++;
               X[i]++;
           }
           else {
               i = 0;
               break;
           }
        }
    }
    return(1);
}
```

```
int sum_E_work()
/*********************************************************
function:
   find the sum (truncated of all minterms in E_work
*********************************************************/
{
    register   i,j;
    int X[MAX_VAR+2];
    int *V;
    int    result;

    for (i=0; i < nvar; i++) X[i] = 0;
    result = 0;
    for (i=0; i < nvar;) {
        V = eval(&E_work,X);
        result = result + V[EVAL];
        X[i]++;
        for (;i < nvar;) {
           if (X[i] >= radix) {
               X[i] = 0;
               i++;
               X[i]++;
           }
           else {
               i = 0;
               break;
           }
        }
    }
    return(result);
}


int minterms()
/*********************************************************
function:
   find the number of minterms in E_work
*********************************************************/

{
    register   i,j;
    int X[MAX_VAR+2];
    int *V;
    int    result;

    for (i=0; i < nvar; i++) X[i] = 0;
    result = 0;
    for (i=0; i < nvar;) {
```

```
        V = eval(&E_work,X);
        result = result + (V[EVAL] >= 1);
        X[i]++;
        for (;i < nvar;) {
           if (X[i] >= radix) {
               X[i] = 0;
               i++;
               X[i]++;
           }
           else {
               i = 0;
               break;
           }
        }
    }
    return(result);
}

int     over_sum_E_work()
/*************************************************
function:
   find the sum (not truncated of all minterms in E_work
*************************************************/
{
    int i,j,result,temp;

    result = 0;
    for(i = 0; i < E_work.nterm; i++) {
        temp = E_work.I[i].coeff;
        for (j = 0; j < nvar; j++ )
           temp = temp * (E_work.I[i].B[j].upper -
               E_work.I[i].B[j].lower +1);
        result = result + temp;
    }
    return(result);
}
```

2. C code for operation control:

```
/* $Reshape.c $
 * $Revision: 2.0 $
 * $Date: 91/09/05  23:37:28 $
 * $Author: Earle and Dueck $
 */

/*
   -This module controls the operations being conducted on the
implicants. It is set up to handle the requirements for both
heuristics.
 */

#include "defs.h"

long a=100001;
int valid;
extern double enumber[MAX_TAB];
extern int R_flag;

 make_a_move()
/*****************************************************
function:
   randomly select two adjacent implicants
*****************************************************/
{
    int success,temp1,temp2,nterm,j,count,i;
    double d_random();

    success = 0;
    count = 0;
    while ((success == 0) && (count < 100)) {
        temp1 = random(E_work.nterm);
        temp2 = random(E_work.nterm);
        while (temp1 == temp2) {
           temp2 = random(E_work.nterm);
        }

        if ((temp1 < 0) || (temp1 >= E_work.nterm) ||
           (temp2 < 0) || (temp2 >= E_work.nterm))
           printf("alarm!! %d %d\n",temp1,temp2);
        if (IsAdj(&E_work.I[temp1],&E_work.I[temp2])){
           success = 1;
           if (combine(&E_work.I[temp1],&E_work.I[temp2])<0)
               c_subtract(temp2);
        }
        else
```

```
            count++;
    }
    return(0);
}


IsAdj(impl,imp2)
/*************************************************************
function:
   test for adjacency
*************************************************************/
Implicant *impl,*imp2;
{
    int used,i,bprime,aprime;

    used = 0;
    for (i = 0; i <  nvar;) {
        aprime = max((impl->B[i].lower),(imp2->B[i].lower));
        bprime = min((impl->B[i].upper),(imp2->B[i].upper));
        if (bprime >= aprime)
           i++;
        else if ((aprime == (bprime + 1)) && (used !=1)) {
           i++;
           used = 1;
        }
        else
              return(0);
    }
return(1);
}

double d_random ()
/*************************************************************
function:
   select random numbers between 0 and nterm
*************************************************************/
{
    a = (a*125)%2796203; /*random number generator */
    return((double)a/2796203);  /*truncate excess*/
}


random (nterm)
    int nterm;
{
    double d_random();
    return((int)(d_random()*nterm));
}
```

```
combine(imp1,imp2)
/*********************************************************
function:
   combine two implicants if possible and to randomly chose
one       of the implicants to be randomly divided.
*********************************************************/
    Implicant *imp1,*imp2;
{
    int j,cost;
    extern double a_temp;

    cost = 0;
    if (IsAbsorb(imp1,imp2)) {
        cost--;
        valid++;
    }
    else if (IsAbsorb(imp2,imp1)) {
        cost--;
        CopyImp(imp1,imp2);
        valid++;
    }
    else if (IsOverlap(imp1,imp2)) {
        imp1->coeff = min(imp1->coeff+imp2->coeff,radix-1);
        cost--;
        valid++;
    }
    else if (IsCombine(imp1,imp2)) {
        for (j=0; j< nvar; j++) {
           imp1->B[j].lower =
                  min(imp2->B[j].lower,imp1->B[j].lower);
           imp1->B[j].upper =
                  max(imp2->B[j].upper,imp1->B[j].upper);
        }
        cost--;
        valid++;
    }
    else if (R_flag) {
        cost = ReshapeCost(imp1,imp2);
        if (d_random() < enumber[cost]) {
           Reshape(imp1,imp2);
           valid++;
           }
        }
    else if (d_random() < enumber[1]) {
        if (random(3) - 1 == 0)
           divide(imp1);
        else
```

```
            divide(imp2);
        cost++;
        valid++;
    }
    return(cost);
}

IsCombine(imp1,imp2)
/*************************************************************
function:
   determine if combinable.
*************************************************************/
    Implicant *imp1,*imp2;
{
    int i,bprime,aprime;
    int used = 0;

    if (imp1->coeff != imp2->coeff)
        return(0);
    for (i = 0; i <  nvar;) {
        aprime = max(imp1->B[i].lower,imp2->B[i].lower);
        bprime = min(imp1->B[i].upper,imp2->B[i].upper);

        if ((imp1->B[i].lower == imp2->B[i].lower) &&
           (imp1->B[i].upper == imp2->B[i].upper)) {
           i++;
        }
        else if ((aprime == (bprime + 1)) && (used !=1)) {
           i++;
           used = 1;
        }
        else
           i = nvar + 5;
    }
    return(i ==nvar);
}

IsOverlap(imp1,imp2)
/*************************************************************
function:
   determine if complete overlap occurs
*************************************************************/
    Implicant *imp1,*imp2;
{
    int i;

    for (i = 0; i <  nvar;) {
        if ((((*imp1).B[i].lower)==((*imp2).B[i].lower))&&
```

```
            (((*imp1).B[i].upper)==((*imp2).B[i].upper)))
            i++;
        else
            return(0);
    }
    return(1);
}

cut(imp,c_var,b_cut)
/*********************************************************
function:
   do a simple divide on an implicant by variable
*********************************************************/
    Implicant *imp;
    int c_var,b_cut;
{
    int old_up;

    old_up = (*imp).B[c_var].upper;
    (*imp).B[c_var].upper = b_cut;
    add_implicant(imp);

    b_cut++;
    E_work.I[E_work.nterm-1].B[c_var].lower = b_cut;
    E_work.I[E_work.nterm-1].B[c_var].upper = old_up;
}

cutcoeff(imp,c_cut_low,c_cut_high)
/*********************************************************
function:
   do a simple divide on an implicant by coefficient
*********************************************************/
    Implicant *imp;
    int c_cut_low,c_cut_high;
{
    imp->coeff = c_cut_low;
    add_implicant(imp);
    E_work.I[E_work.nterm-1].coeff = c_cut_high;
}


divide(Imp)
/*********************************************************
function:
   randomly divide an implicant
*********************************************************/
    Implicant *Imp;
    {
```

```
int i,total_cuts,r_cut,c_count;
int k, kprime, j, list1[20], list2[20];

total_cuts = 0;
for(i = 0; i < nvar; i++) {
    total_cuts = total_cuts +
              Imp->B[i].upper - Imp->B[i].lower;
}

if (Imp->coeff == radix - 1) {
    i=1;
    for (k=1; k<radix; k++) {
       if (k>Imp->coeff - k) {
           kprime = k;
       }
       else {
           kprime = (Imp->coeff - k);
       }
       for (j= kprime; j<radix; j++) {
           list1[i] = k;
           list2[i] = j;
           i++;
       }
    }
    c_count = i-1;
}
else {
    c_count = Imp->coeff/2;
}
total_cuts = total_cuts + c_count;

if (total_cuts != 0) {
    r_cut = random (total_cuts) + 1;
    if ((Imp->coeff == radix-1)&&(r_cut <= c_count)) {
       cutcoeff(Imp,list1[r_cut],list2[r_cut]);
    }
    else if (r_cut < Imp->coeff) {
       cutcoeff(Imp,r_cut,Imp->coeff - r_cut);
    }
    else {
       r_cut = r_cut - c_count;
       i=0;
       while ((Imp->B[i].upper - Imp->B[i].lower) <
                  r_cut) {
             r_cut = r_cut - (Imp->B[i].upper -
                         Imp->B[i].lower);
           i++;
       }
```

```
            r_cut = Imp->B[i].lower + (r_cut-1);
            cut(Imp,i,r_cut);
        }
    }
}


IsAbsorb(imp1,imp2)
/*************************************************************
function:
   determine if one implicant can absorb another
*************************************************************/
    Implicant *imp1,*imp2;
{
    int i;
    i = 0;

    if(imp1->coeff != (radix - 1))
        return(0);
    for (i = 0; i <  nvar;) {
        if ((imp1->B[i].lower <= imp2->B[i].lower) &&
            (imp1->B[i].upper >= imp2->B[i].upper)) {
            i++;
        }
        else {
            return(0);
        }
    }
    return(1);
}

PrintImp(I)
/*************************************************************
function:
   print the implicant
*************************************************************/
    Implicant *I;
{
    int i; printf("+%d",(*I).coeff);
    for (i = 0; i < nvar; i++)
    {
    printf("*x%d(%d,%d)",i+1,(*I).B[i].lower,(*I).B[i].upper);
    }
    printf("\n");}
```

LIST OF REFERENCES

1. Pomper, G. and Armstrong, J. A., "Representation of multi-valued functions using the direct cover method," *IEEE Transactions on Computing*, September 1981, pp. 674-679.

2. Dueck, G. W. and Miller, D. M., "A direct cover MVL minimization using the truncated sum," *Proceedings of the 17th International Symposium for Multiple-Valued Logic*, May 1987, pp. 221-227.

3. Yang C. and Wang, Y. M., "A neighborhood decouping algorithm for truncated sum minimization," *Proceeding of the 20th International Symposium on Multiple-Valued Logic*, May 1990, pp. 153-160.

4. Kirkpatrick, S., Gelatt, C. D. Jr., and Vecchi, M. P., "Optimization by simulated annealing," *Science*, vol. 220, No. 4598, 13 May 1983, pp. 671-680.

5. Dueck, G. W., Earle, R. C., Tirumalai, P., and Butler, J. T., "Multiple-Valued Programmable Logic Array Minimization by Simulated Annealing," Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA, October 1991.

6. Yurchak, J. and Butler, J. T., "HAMLET - An expression compiler/optimizer for the implementation of heuristics to minimize multiple-valued programmable logic arrays," *Proceedings of the 20th International Symposium on Multiple-valued Logic*, May 1990, pp. 144-152.

7. Anton, H. and Rorres, C., *Elementary Linear Algebra with Applications*, John Wiley & Sons Inc., 1987.

8. Green, J. W. and Supowit, K. J., "Simulated Annealing Without Rejected Moves," *IEEE Transactions on Computer-Aided Design*, vol. CAD-5, no. 1, January 1986, pp. 221-228.

9. Besslich, P. W., "Heuristic minimization of MVL functions: A direct cover approach," *IEEE Transactions on Computing*, February 1986, pp. 134-144.

10. Brayton, R.K., Hachtel, G. D., McMullen, C. T., and Sangiovanni-Vincentelli, A., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic, Boston 1984.

11. Isaacson, D. L. and Madsen, R. W., *Markov chains: theory and applications*, John Wiley & Sons Inc., 1976.

INITIAL DISTRIBUTION LIST

		No. of Copies
1.	Defense Technical Information Center Cameron station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor Jon T. Butler, Code EC/Bu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
5.	Professor Chyan Yang, Code EC/Ya Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Dr. Gerhard Dueck, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
7.	Lt. Robert C. Earle P.O. Box 518 Farmerville, LA 71241	1

8. Dr. George Abraham, Code 1005 — 1
Office of Research and Technology
Naval Research Laboratories
4555 Overlook Ave., N.W.
Washington, DC 20375

9. Dr. Robert Williams — 1
Naval Air Development Center, Code 5005
Warminister, PA 18974-5000

10. Dr. James Gault — 1
U.S. Army research Office
P.O. Box 12211
Research Triangle Park, NC 27709

11. C. Lee Giles — 1
AFOSR, Bldg. 410
Bolling, AFB, DC 20332

12. Dr. Andre van Tilborg — 1
Office of Naval Research
Code 1133
800 N. Quincy Str.
Arlington, VA 22217-5000

13. Dr. Clifford Lau — 1
Office of Naval Research
1030 E. Green Str.
Pasadena, CA 91106-2485